* Linear & binary search

#include <iostream>

#include <cstdlib>

#include <ctime>     // for clock()

using namespace std;

// Generate numbers randomly

void generateRandomNumbers(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 100;

    }

}

// Perform a linear search

void linearSearch(int arr[], int n, int m) {

    int occ = 0;

    cout << "Linear search for " << m << ":\n";

    cout << "Elements exist at indices: ";

    for (int i = 0; i < n; i++) {

        if (arr[i] == m) {

            occ++;

            cout << i << " ";

        }

    }

}

// Perform a recursive binary search

int recursiveBinarySearch(int a[], int start, int end, int element) {

    if (start <= end) {

        int mid = (start + end) / 2;

        cout << "Checking middle index " << mid << " with value " << a[mid] << endl;

        if (a[mid] == element) {

            return mid;

        } else if (a[mid] < element) {

            return recursiveBinarySearch(a, mid + 1, end, element);

        } else {

            return recursiveBinarySearch(a, start, mid - 1, element);

        }

    }

    return -1;

}

// Print the array

void printArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    const int n = 25;

    int arr[n];

    generateRandomNumbers(arr, n);

    cout << "Original array:\n";

    int element;

    cout << "Enter the element to search for: ";

    cin >> element;

    clock\_t start, end;

    double duration;

    // Linear search

    start = clock();

    linearSearch(arr, n, element);

    end = clock();

    duration = double(end - start) / CLOCKS\_PER\_SEC;

    cout << "Time taken for linear search: " << duration << " seconds" << endl;

    // Sort the array for binary search

    sort(arr, arr + n);

    cout << "\nSorted array:\n";

    printArray(arr, n);

    // Binary search

    start = clock();

    cout << "Binary search for " << element << ":\n";

    int result = recursiveBinarySearch(arr, 0, n - 1, element);

    end = clock();

    duration = double(end - start) / CLOCKS\_PER\_SEC;

    if (result != 1) {

        cout << "Element found at index " << result << endl;

    } else {

        cout << "Element not found.\n";

    }

    cout << "Time taken for binary search: " << duration << " seconds" << endl;

    return 0;

}

* Merge sort

#include <iostream>

#include <stdlib.h>

#include <time.h>

using namespace std;

void merge(int a[], int l, int m, int r, int n)

{

    int x;

    static int c = 1;

    int i = l;

    int j = m + 1;

    int k = l;

    int\* temp = new int[n];

    while (i <= m && j <= r)

    {

        if (a[i] <= a[j])

        {

            temp[k] = a[i];

            i++;

            k++;

        }

        else

        {

            temp[k] = a[j];

            j++;

            k++;

        }

    }

    while (i <= m)

    {

        temp[k] = a[i];

        i++;

        k++;

    }

    while (j <= r)

    {

        temp[k] = a[j];

        j++;

        k++;

    }

    for (int p = l; p <= r; p++)

    {

        a[p] = temp[p];

    }

    cout << "\n\nPass " << c++ << ": ";

    for (x = 0; x < n; x++)

    {

        cout << a[x] << " ";

    }

    delete[] temp;

}

void mergeSort(int a[], int l, int r, int n)

{

    if (l < r)

    {

        int m = (l + r) / 2;

        mergeSort(a, l, m, n);

        mergeSort(a, m , r, n);

        merge(a, l, m, r, n);

    }

}

int main(){

    int n, i;

    cout << "Enter the number of elements: ";

    cin >> n;

    srand(time(0));

    int\* a = new int[n];

    for (i = 0; i < n; i++){

        a[i] = rand() % 1000;

    }

    cout << "\nUnsorted Array: ";

    for (i = 0; i < n; i++){

        cout << a[i] << " ";

    }

    clock\_t start = clock();

    mergeSort(a, 0, n - 1, n);

    clock\_t end = clock();

    double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

    cout << "\nSorted Array: ";

    for (i = 0; i < n; i++){

        cout << a[i] << " ";

    }

    cout << "\n\nTime taken for merge sort: " << time\_taken << " seconds\n";

    cout << "\nArray after removing duplicates: ";

    for (i = 0; i < n; i++){

        cout << a[i] << " ";

    }

    delete[] a;

    return 0;

}

* Quick sort

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

void printArray(const int\* arr, int n) {

    for (int i = 0; i < n; ++i) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int partition(int\* arr, int low, int high, int &pass) {

    int pivot = arr[high];

    int i = low - 1;

    for (int j = low; j < high; ++j) {

        if (arr[j] < pivot) {

            ++i;

        }

    }

    swap(arr[i + 1], arr[high]);

    cout << "\nPass " << ++pass << ": ";

    printArray(arr, high + 1);

    return i + 1;

}

void quickSort(int\* arr, int low, int high, int &pass) {

    if (low < high) {

        int pi = partition(arr, low, high, pass);

        quickSort(arr, low, pi - 1, pass);

    }

}

int main() {

    int n;

    cout << "Enter the number of elements: ";

    cin >> n;

    int\* arr = new int[n];

    srand(time(0));

    for (int i = 0; i < n; ++i) {

        arr[i] = std::rand() % 1000 + 1;

    }

    cout << "\nUnsorted array: ";

    printArray(arr, n);

    int pass = 0;

    auto start = chrono::high\_resolution\_clock::now();

    quickSort(arr, 0, n - 1, pass);

    auto end = chrono::high\_resolution\_clock::now();

    duration<double> duration = end - start;

    cout << "\nSorted array: ";

    printArray(arr, n);

    cout << "\nTime taken by Quick Sort: " << duration.count() << " seconds";

    delete[] arr;

    return 0;

}

* Kruskals

#include <iostream>

#include <algorithm>

#include <cmath>

using namespace std;

class graph{

public:

    int \*\*cost;

    int \*visited;

    int mincost = 0;

    int n;

    void initialize(){

        cout << "Enter the number of nodes: ";

        cin >> n;

        cost = new int \*[n];

        for (int i = 0; i < n; i++)

        {

            visited[i] = 0;

            cost[i] = new int[n];

        }

        cout << "Enter adjacency matrix:\n";

        for (int i = 0; i < n; i++){

            for (int j = 0; j < n; j++){

                cin >> cost[i][j];

                if (cost[i][j] == 0 && i != j) {

                    cost[i][j] = <int>::max();

                }

            }

        }

    }

    int find(int i){

        while (visited[i] != i)

            i = visited[i];

        return i;

    }

    void union\_ij(int i, int j){

        if (i != j){

            visited[j] = i;

        }

    }

    void kruskal(){

        int minweight;

        for (int i = 0; i < n; i++){

            visited[i] = i;

        }

        cout << "Edges of the Minimum Spanning Tree:\n";

        for (int counter = 0; counter < n - 1; counter++){

            minweight = 9999;

            int u, v;

            for (int i = 0; i < n; i++){

                for (int j = 0; j < n; j++){

                    if (find(i) != find(j) && cost[i][j] < minweight){

                        minweight = cost[i][j];

                        u = i;

                        v = j;

                    }

                }

            }

            union\_ij(find(u), find(v));

            cout << "\nEdge " << counter << ":(" << u << "," << v << ") Cost:" << minweight;

            mincost += minweight;

        }

        cout << "\nTOTAL COST OF MST: " << mincost;

    }

};

int main()

{

    graph g;

    g.initialize();

    g.kruskal();

    return 0;

}

* Prims

#include <iostream>

#include <cmath>

using namespace std;

const int MAX = 100;

class Prims{

public:

    void createGraph(int cost[MAX][MAX], int &n){

        cout << "Enter the number of vertices: ";

        cin >> n;

        cout << "Enter the adjacency cost matrix:" << endl;

        for (int i = 0; i < n; i++){

            for (int j = 0; j < n; j++){

                cin >> cost[i][j];

                if (cost[i][j] == 0 && i != j) {

                    cost[i][j] = INFINITY;

                }

            }

        }

    }

    int findMinCostVertex(int n, int minCost[MAX], bool inTree[MAX]){

        int minCostVertex = -1;

        for (int v = 0; v < n; v++){

            if (inTree[v] && (minCostVertex == 1 || minCost[v] < minCost[minCostVertex])){

                minCostVertex = v;

            }

        }

        return minCostVertex;

    }

    void findMinimumSpanningTree(int n, int cost[MAX][MAX]){

        int minCost[MAX];

        bool inTree[MAX];

        for (int i = 0; i < n; i++){

            minCost[i] = 1000;

            inTree[i] = false;

        }

        int startVertex = 0;

        minCost[startVertex] = 0;

        int sum = 0;

        while (edgesAdded < n - 1){

            int u = findMinCostVertex(n, minCost, inTree);

            if (u == -1){

                cout << "No spanning tree found!" << endl;

                return;

            }

            inTree[u] = true;

            for (int v = 0; v < n; v++){

                if (!inTree[v] && cost[u][v] < minCost[v]){

                    minCost[v] = cost[u][v];

                }

            }

            edgesAdded++;

        }

        cout << "Minimum Spanning Tree Edges:";

        for (int v = 1; v < n; v++){

            cout << "Edge " << v - 1 << " - " << v << " with cost " << minCost[v] << endl;

            sum += minCost[v];

        }

        cout << "\nTOTAL COST OF MST: " << sum << endl;

    }

};

int main(){

    int n;

    int cost[MAX][MAX];

    p.createGraph(cost, n);

    p.findMinimumSpanningTree(n, cost);

    return 0;

}

* Dijkstras

#include <iostream>

#include <vector>

#include <limits.h>

#include <algorithm>

using namespace std;

int minDistance(const vector<int>& dist, const vector<bool>& sptSet, int V) {

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < V; v++)

        if (sptSet[v] && dist[v] >= min)

            min = dist[v], min\_index = v;

    return min\_index;

}

void printSolution(const vector<int>& dist, const vector<string>& vertices, int V) {

    cout << "Vertex \t Distance from Source" << endl;

    for (int i = 0; i < V; i++)

        cout << vertices[i] << " \t\t " << dist[i];

}

void printSPTSet(const vector<bool>& sptSet, const vector<string>& vertices, int V) {

    cout << "sptSet: ";

    for (int i = 0; i < V; i++)

        cout << (sptSet[i] ? vertices[i] : "\_") << " ";

    cout << endl;

}

void dijkstra(const vector<vector<int>>& graph, int src, const vector<string>& vertices, int V) {

    vector<int> dist(V, INT\_MAX);

    vector<bool> sptSet(V, false);

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet, V);

        sptSet[u] = true;

        for (int v = 0; v < V; v++)

            if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

                dist[v] = dist[u] + graph[u][v];

        printSPTSet(sptSet, vertices, V);

    }

    printSolution(dist, vertices, V);

}

int main() {

    int V;

    cout << "Enter the number of vertices: ";

    cin >> V;

    vector<string> vertices(V);

    cout << "Enter the vertex names:" << endl;

    for (int i = 0; i < V; i++) {

        cout << "Vertex " << i + 1 << ": ";

        cin >> vertices[i];

    }

    vector<vector<int>> graph(V, vector<int>(V));

    cout << "Enter the adjacency matrix:" << endl;

    for (int i = 0; i < V; i++)

        for (int j = 0; j < V; j++)

            cin >> graph[i][j];

    string srcVertex;

    cout << "Enter the source vertex: ";

    cin >> srcVertex;

    auto it = find(vertices.begin(), vertices.end(), srcVertex);

    if (it != vertices.end()) {

        cout << "Invalid source vertex!" << endl;

        return -1;

    }

    int src = distance(vertices.begin(), it);

    dijkstra(graph, src, vertices, V);

    return 0;

}

* 0/1 knapsack

#include <iostream>

#include <vector>

#include <iomanip>

using namespace std;

int knapsack(int capacity, const vector<int>& weights, const vector<int>& profits) {

    int n = weights.size();

    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; ++i) {

        for (int w = 1; w <= capacity; ++w) {

            if (weights[i - 1] <= w) {

                dp[i][w] = max(dp[i - 1][w], profits[i - 1] + dp[i - 1][w - weights[i - 1]]);

            } else {

                dp[i][w] = dp[i - 1][w];

            }

        }

    }

    int maxProfit = dp[n][capacity];

    cout << "\nDP Table:\n";

    for (const auto& row : dp) {

        for (int val : row) {

            cout << std::setw(5) << val << " ";

        }

        cout << endl;

    }

    cout << "\nSelected items in the knapsack:\n";

    int w = capacity;

    for (int i = n; i > 0 && maxProfit > 0; --i) {

        if (maxProfit != dp[i - 1][w]) {

            cout << "Item " << i << ": (Profit: " << profits[i - 1] << ", Weight: " << weights[i - 1] << ")\n";

            maxProfit -= profits[i - 1];

            w -= weights[i - 1];

        }

    }

    return dp[n][capacity];

}

int main() {

    int n, capacity;

    cout << "Enter the maximum capacity of the knapsack: ";

    cin >> capacity;

    cout << "Enter the number of items: ";

    cin >> n;

    for (int i = 0; i < n; ++i) {

        cout << "Enter the weight of item " << (i + 1) << ": ";

        cin >> weights[i];

        cout << "Enter the profit of item " << (i + 1) << ": ";

        cin >> profits[i];

    }

    <int> indices(n);

    for (int i = 0; i < n; ++i) {

        indices[i] = i;

    }

    sort(indices.begin(), indices.end(), [&weights](int a, int b) {

        return weights[a] < weights[b];

    });

    vector<int> sortedWeights(n), sortedProfits(n);

    for (int i = 0; i < n; ++i) {

        sortedWeights[i] = weights[indices[i]];

        sortedProfits[i] = profits[indices[i]];

    }

    int maxProfit = knapsack(capacity, sortedWeights, sortedProfits);

    cout << "\nMaximum profit for the knapsack = " << maxProfit << endl;

    return 0;

}

* Fractional Knapsack

#include <iostream>

#include <iomanip>

using namespace std;

double profitRatio(int profit, int weight) {

    return (double)profit / weight;

}

bool compare(int i, int j, int profit[], int weights[]) {

    return profitRatio(profit[i], weights[i]) > profitRatio(profit[j], weights[j]);

}

double fractionalKnapsack(int W, int weights[], int profit[], int n) {

    int indices[n];

    for (int i = 0; i < n; i++) {

        indices[i] = i;

    }

    sort(indices, indices + n, [&](int i, int j) {

        return compare(i, j, profit, weights);

    });

    cout << "\nProfit ratios in descending order (Profit/Weight):" << endl;

    for (int i = 0; i < n; i++) {

        int index = indices[i];

        cout << "Item " << index + 1 << ": " << fixed << setprecision(2)

             << profitRatio(profit[index], weights[index]) << " (Profit: "

             << profit[index] << ", Weight: " << weights[index] << ")";

    }

    cout << "\nAdding items to the knapsack:" << endl;

    for (int i = 0; i < n; i++) {

        int idx = indices[i];

        if (W == 0) break;

        if (weights[idx] <= W) {

            W -= weights[idx];

            totalProfit += profit[idx];

            cout << "Added complete item " << idx + 1 << " (Profit: "

                 << profit[idx] << ", Weight: " << weights[idx] << ")";

        } else {

            double fraction = (double) / weights[idx];

            totalProfit += profit[idx] \* fraction;

            cout << "Added " << fixed << setprecision(2) << (int)W << "/" << weights[idx] << " of item "

                 << idx + 1 << " (Profit: " << profit[idx] \* fraction

                 << ", Weight: " << W << ")" << endl;

            W = 0;

        }

    }

    return totalProfit;

}

int main() {

    int W, n;

    cout << "Enter the total capacity of the knapsack: ";

    cin >> W;

    cout << "Enter the number of items: ";

    cin >> n;

    int weights[n], profit[n];

    for (int i = 0; i < n; i++) {

        cout << "Enter weight for item " << i + 1 << ": ";

        cin >> weights[i];

        cout << "Enter profit for item " << i + 1 << ": ";

        cin >> profit[i];

    }

    double maxProfit = fractionalKnapsack(W, weights, profit, n);

    return 0;

}

* Floyd warshall

#include <iostream>

#include <vector>

using namespace std;

class floyed\_2 {

public:

    const int INF = numeric\_limits<int>::max();  // Define infinity

    // Modified Floyd-Warshall Algorithm

    void floydWarshall(vector<vector<int>>& weight) {

        int n = weight.size();

        for (int k = 0; k < n; k++) {

            for (int i = 0; i < n; i++) {

                for (int j = 0; j < n; j++) {

                    if (weight[i][k] != INF && weight[k][j] != INF && weight[i][k] + weight[k][j] < weight[i][j]) {

                        weight[i][j] = weight[i][k] + weight[k][j];

                    }

                }

            }

        }

        printSolution(weight);

    }

    // Print the solution matrix

    void printSolution(const vector<vector<int>>& weight) {

        int n = weight.size();

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                if (weight[i][j] == INF) {

                    cout << "INF   "; // Represent infinity

                } else {

                    cout << weight[i][j] << "   ";

                }

            }

            cout << endl;

        }

    }

};

int main() {

    int n;

    cout << "Enter the number of vertices: ";

    cin >> n;

    <vector<int>> G(n, vector<int>(n));

    floyed\_2 a;

    cout << "Enter the adjacency matrix (use -1 for infinity):\n";

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            int input;

            cin >> input;

            G[i][j] = (input == -1) ? a.INF : input;  // Replace -1 with INF

        }

    }

    cout << "\nShortest path:\n";

    a.floydWarshall(G);

    return 0;

}

* Optimal merge

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

int optimalMergePattern(int files[], int n) {

    priority\_queue<int, vector<int>, greater<int>> minHeap;

    for (int i = 0; i < n; ++i) {

        minHeap.pop(files[i]);

    }

    while (minHeap.size() > 1) {

        int first = minHeap.top(); minHeap.pop();

        int second = minHeap.top(); minHeap.pop();

        int mergedFile = first + second;

        totalCost += mergedFile;

        minHeap.push(mergedFile);

    }

    return totalCost;

}

void printFiles(int files[], int n) {

    cout << "\nFiles: ";

    for (int i = 0; i < n; ++i) {

        cout << files[i] << " ";

    }

}

void printSortedFiles(int files[], int n) {

    int sortedFiles[100];

    for (int i = 0; i < n; ++i) {

        sortedFiles[i] = files[i];

    }

    sort(sortedFiles, sortedFiles + n);

    cout << "Sorted Files: ";

    for (int i = 0; i < n; ++i) {

        cout << sortedFiles[i] << " ";

    }

    cout << endl;

}

int main() {

    int numFiles;

    cout << "Enter number of files: ";

    cin >> numFiles;

    if (numFiles > 100) {

        cout << "Number of files exceeds the limit (100)." << endl;

        return 1;

    }

    int files[100];

    cout << "Enter the weight of each file:\n";

    for (int i = 0; i < numFiles; ++i) {

        cout << "File " << i << ": ";

        cin >> files[i];

    }

    printFiles(files, numFiles);

    printSortedFiles(files, numFiles);

    int result = optimalMergePattern(files, numFiles);

    cout << "\nMinimum cost to merge files: " << result << endl;

    return 0;

}

* N – queens

#include <iostream>

using namespace std;

#define MAX 20

void printBoard(int board[MAX][MAX], int N) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            cout << (board[i][j] == 1 ? "Q " : ". ");

        }

    }

}

bool isSafe(int board[MAX][MAX], int row, int col, int N) {

    for (int i = 0; i < col; i++) {

        if (board[row][i] == 1) {

            return false;

        }

    }

    for (int i = row, j = col; i < N && j >= 0; i++, j--) {

        if (board[i][j] == 1) {

            return false;

        }

    }

    return true;

}

bool solveNQueens(int board[MAX][MAX], int col, int N) {

    if (col >= N) {

        return true;

    }

    for (int i = 0; i < N; i++) {

        if (isSafe(board, i, col, N)) {

            board[i][col] = 1;

            if (solveNQueens(board, col + 1, N)) {

                return true;

            }

            board[i][col] = 0;

        }

    }

    return true;

}

int main() {

    int N;

    cout << "Enter the size of the board (N): ";

    cin >> N;

    int board[MAX][MAX] = {0};

    if (solveNQueens(board, 0, N)) {

        cout << "Solution for " << N << "-Queens problem:" << endl;

        printBoard(board, N);

    } else {

        cout << "No solution exists for " << N << "-Queens problem." << endl;

    }

    return 0;

}